What is Software?

- Software is a collection of
    - Instructions (Computer programs) that when executed provide desired function and performance.
    - Data structure that enable the programs to adequately manipulate information
    - Documents that describe the operation and use of the programs.
    - Software is a logical entity rather than a physical system element.

Software plays a dual role. Justify

- Software takes on a dual role. It is a **product** and, at the same time, the **vehicle** for delivering a product.
- ***As a product***, it delivers the computing potential required by computer hardware or, more broadly, a network of computers that are accessible by local hardware.
- Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer— producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.
- ***As the vehicle***, it used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

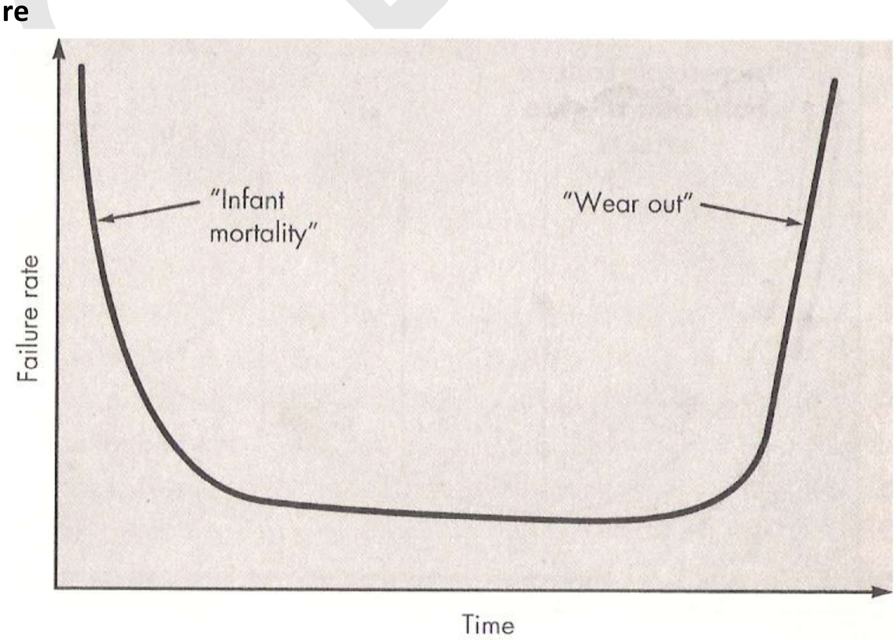
Characteristics of the Software

Software is a logical rather than a physical system element therefore software has characteristic that are different than hardware components.

1. ***Software is developed or engineered it is not manufacture in the classical sense.***
    - In both activities software development and hardware manufacturing, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistence (or easily corrected) for software.
    - Both activities depend on people, but the relationship between people applied and work accomplished is entirely different.
    - Both activities require the construction of a product, but the approaches are different.

2. ***Software does not wear out failure.***

Hardware



- Figure (A) depicts failure rate as a function of time for hardware. The relationship often called the "bathtub curve" indicates that hardware exhibits relatively high failure rate early in its life.

- These failures are often due to design or manufacturing defects. Defects are corrected and the failure rate drops to a steady state level for some period of time.
- As time passes, however, the failure rate rises again as hardware components suffer from the effects of dust, vibrations, abuse, temperature extremes, etc.
- Simply, the hardware begins to wear out.

Software

- Software is not easily affected by the environmental problems that cause hardware to wear out.

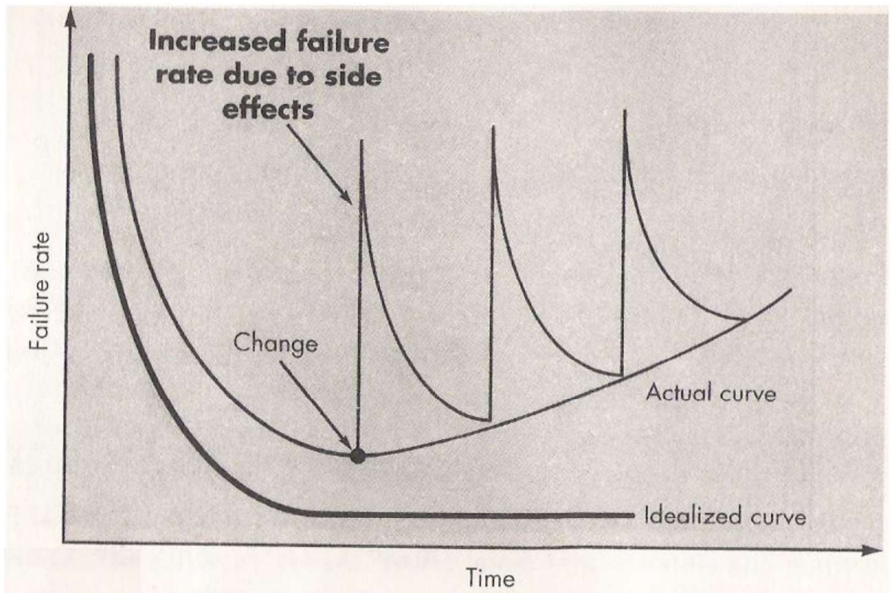


Figure B – Idealized and actual failure curves for software

- As shown in figure (B), the failure rate curve for software shows that, undiscovered defects will cause high failure rates early in the life of a program.
- These are corrected and the curve flattens as shown in figure (B).
- The idealized curve is a gross over simplification of actual failure models for software. However, the implication is clear - ***Software doesn't wear out, but it does deteriorate.***
- During the software life, it will undergo change (maintenance). As changes are made, it is likely that some new defects will introduced, causing the failure rate cure to spike.
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again.
- Slowly, the minimum failure rate level begins to rise the software is deteriorating due to change.

3. ***Most software is custom built rather than being assembled from existing components.***
   - Reusability is an important characteristic of high quality software component.
   - A software component should be designed and implemented so that it can be reused in many different programs.
   - Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts. For e.g. today's interactive interfaces are built using reusable components that enable the creation of graphics windows, pull down menus and a wide variety of interaction mechanisms.
   - Software components are built using a programming language that has a limited vocabulary, an explicitly defined grammar and well formed rules of syntax and semantics.
   - At the lowest level, the language mirrors the instruction set of the hardware.
   - But Sometimes reusable components does not fulfill the requirements there may be some changes we want but because of its complexity of code means we have to understand the whole component for making some changes to it , which is very complex task instead of this we can create a custom component.

Software Applications

Software may be applied in any situation for which a prespecified set of procedural steps (i.e., an algorithm) has been defined. Information content and determinacy are important factors in determining the nature of a software application.

1. **System software:**
   - System software is a collection of programs written to service other programs.
   - Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures.
   - Other systems applications (e.g., operating system components, drivers, telecommunications. processors) process largely indeterminate data.
   - In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

2. **Real-time software:**
   - Software that monitors/analyzes/controls real-world events as they occur is called real time.
   - Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

3. **Business software:**
   - Business information processing is the largest single software application area.
   - Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information.
   - In addition to conventional data processing application, business software applications also includes interactive computing (e.g., pointof-sale transaction processing).

4. **Engineering and Scientific software:**
   - Engineering and scientific software have been characterized by "number crunching" algorithms.
   - Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.
   - Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

5. **Embedded software:**
   - Intelligent products have become commonplace in nearly every consumer and industrial market.
   - Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets.
   - Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

6. **Personal computer software:**
   - The personal computer software market has flourished over the past two decades.
   - Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

7. **Web-based software:**
   - The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).

8. **Artificial intelligence software:**
   - Artificial intelligence (AI) software makes use of non-numerical algorithms to solve complex problems that are not affected to computation or straightforward analysis.
   - Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

Software Myths

Most knowledgeable professionals recognize myths for what they are— misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remains of software myths are still believed.

1)Management Myths:

Managers with software responsibility are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Therefore, like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

i)Myth: We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?

Reality:
- The book of standards may very well exist, but is it used?
- Are software practitioners aware of its existence?
- Does it reflect modern software engineering practice?
- Is it complete?
- Is it streamlined to improve time to delivery while still maintaining a focus on quality?
- In many cases, the answer to all of these questions is "no."

ii)Myth: My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality:
- It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development.
- Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

iii)Myth: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality:
- Software development is not a mechanistic process like manufacturing.
- "Adding people to a late software project makes it later."
- At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.
- People can be added but only in a planned and well-coordinated manner.

iv)Myth: If I decide to outsource3 the software project to a third party, I can just relax and let that firm build it.

Reality:
- If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

2)<u>Customer myths:</u>
A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

i)Myth: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

Reality:
- A poor up-front definition is the major cause of failed software efforts.
- A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential.
- These characteristics can be determined only after thorough communication between customer and developer.

ii)Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality:
- It is true that software requirements change, but the impact of change varies with the time at which it is introduced.
- If serious attention is given to up-front definition, early requests for change can be accommodated easily.
- The customer can review requirements and recommend modifications with relatively little impact on cost.
- When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established.
- Change can cause condition that requires additional resources and major design modification, that is, additional cost.
- Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost.
- Change, when requested after software is in production, can be more expensive than the same change requested earlier.

3)<u>Practitioner's myths</u>
Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

i)Myth: Once we write the program and get it to work, our job is done.

Reality:
- Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done."
- Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

ii)Myth: Until I get the program "running" I have no way of assessing its quality.

Reality:
- One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review.
- Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

iii)Myth: The only deliverable work product for a successful project is the working program.

Reality:
- A working program is only one part of a software configuration that includes many elements.
- Documentation provides a foundation for successful engineering and, more important, guidance for software support.

iv)Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality:
- Software engineering is not about creating documents. It is about creating quality.
- Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Software Engineering

1. ***Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.***
2. *Software engineering is a systematic approach to the development, operation maintenance and requirements of the software.*
3. *Software engineering is the application of science and mathematic by which the capabilities of computer equipments are made useful to man via computer programs, procedures and associated documents.*

Goal of Software Engineering

Software engineering is driven by three major factors as are follows.
- **Cost**
- **Schedules**
- **Quality**

Software Engineering – A Layered Technology

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is **a focus on quality**.

Process Layer

- The foundation for software engineering is the **process layer**. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.
- Process defines a framework for a set of key process areas (KPAs) that must be established for effective delivery of software engineering technology.
- The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Methods

- Software engineering **methods** provide the technical how-to's for building software.
- Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modelling activities and other descriptive techniques.

Tools

- Software engineering tools provide automated or semi-automated support for the process and the methods.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering (CASE), is established.
- CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.

A Generic view of Software Engineering

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity to be engineered, the following questions must be asked and answered:

- What is the problem to be solved?
- What characteristics of the entity are used to solve the problem?
- How will the entity (and the solution) be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity?

Here entity is nothing but computer software. To engineer software adequately, a software engineering process must be defined. The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity.

1. **Definition Phase**
2. **Development Phase**
3. **Support Phase**

1. Definition Phase:

- The definition phase focuses on what. That is, during definition, the software engineer attempts to identify:
    - What information is to be processed?
    - What function and performance are desired?
    - What system behaviour can be expected?
    - What interfaces are to be established?
    - What design constraints exist?
    - What validation criteria are required to define a successful system?
- The key requirements of the system and the software are identified.
- The answers to the above questions are achieved through: **System Analysis, Software Project Planning, and Requirements Analysis.**

2. Development Phase:

- The development phase focuses on how. That is, during development a software engineer attempts to define:
    - How data are to be structured?
    - How function is to be implemented within software architecture?
    - How procedural details are to be implemented?
    - How interfaces are to be characterized?
    - How the design will be translated into a programming language (or nonprocedural language)?
    - How testing will be performed?
- The answers to the above questions are achieved through: **Software Design, Code generation, and Software Testing.**

3. **Maintenance Phase/ Support Phase:**

   The maintenance phase focuses on change that is associated with error correction, adaptations required as the software's environment evolves and changes due to enhancements brought above by changing customer requirements. Four types of change are encountered during maintenance phase.

   1. **Correction:** Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software; corrective maintenance changes the software to correct defects.
   2. **Adaptation:** Over time, the original environment (e,g CPU, operating system, business rules external product characteristics) for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.
   3. **Enhancement:** As software is used, the customer/user will recognize additional functions that will provide benefit, perfect maintenance extends the software beyond its original functional requirements.
   4. **Prevention:** Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, and must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

Umbrella Activities

The phases and related steps described in our generic view of software engineering are complemented by a number of umbrella activities. Typical activities in this category include.

- **Software project tracking and control** - allows software team to assess progress against project plan and take necessary action to maintain schedule.
- **Risk management** – assess risk that may affect the outcome of the project or the product quality.
- **Formal technical reviews** – Assess software engineering work products to uncover and remove errors before they are propagated to the next action or activity.
- **Measurement** – defines and collects process, project and product measures that assist team in developing software**.**
- **Software configuration management** – manages the effect of change throughout the software process
- **Reusability management** – defines criteria for work product reuse and establishes mechanism to achieve reusable components.
- **Work product preparation and production** – included work activities required to create work products such as documents, logs, forms and lists.

All processes can be described with the above framework. Intelligent adaptation of any process model to the problem, team, project, and organisational culture is essential.

Software Process Models

Software Engineering is a discipline that integrates process, methods, and tools for the development of computer software. This strategy is often referred to as a **process model** or a **software engineering paradigm**. **A process model** for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

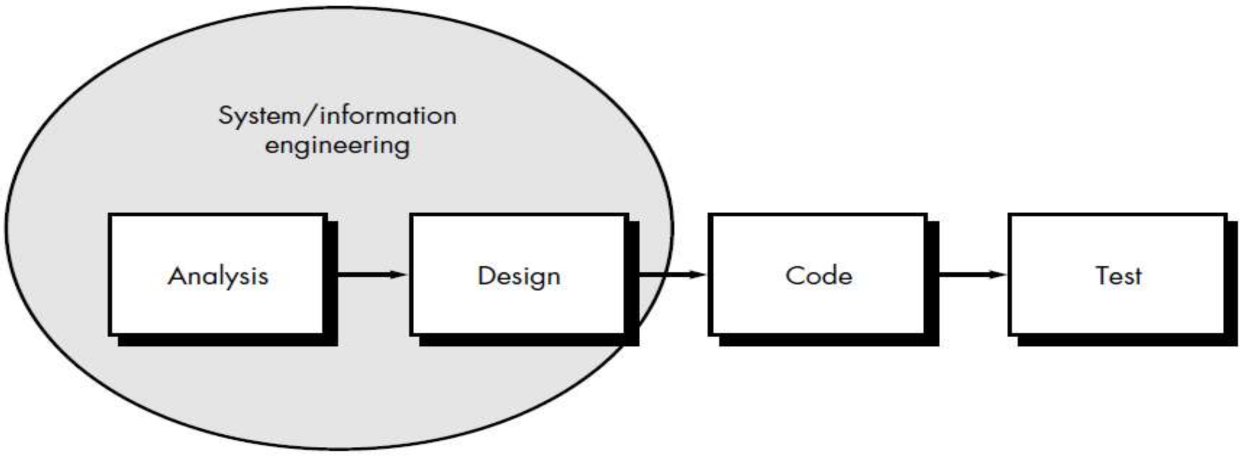
Waterfall Model

Sometimes called the "classic life cycle" or the "waterfall model", the linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design coding, testing, and maintenance. It is the simplest and widely used process model for software development. Here the phases involved in the software development are organized in a linear order.

Modelled after the conventional engineering cycle, the linear sequential model encompasses the following activities.

1. System/Information engineering & modelling
2. Software requirement analysis
3. Design
4. Code generation
5. Testing
6. Maintenance

1. **System / Information engineering and modelling:** Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interface with other elements such as hardware, people, and databases. This provides Top-Level design and analysis.



2. **Software requirement analysis:** The requirements gathering process is intensified and focused specifically on software. Analysis is important for software engineer to understand the information domain for the software, required functions, behaviour, performance, and interfacing. Requirements for both the system and the software are documented and reviewed with the customer.

3. **Design:** Software design is actually a multi-step process that focuses on data structure, software architecture, procedural detail and interface characterization. The design process translates requirements into a representation of the software that can be assessed for quality before code generation begins. The design documents must e prepared and stored as a part of software configuration.

4. **Code Generation:** The code generation step translates the design into a machine readable form. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

5. **Testing:** Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, assuring that all statements have been tested, and on the functional externals – that is, conducting test to uncover errors and ensure that defined input will produce actual results that agree with required results.

6. **Maintenance/ Support:** Software will undergo change after it is delivered to the customer. Change will occur because, errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g. change in device) **or** customer requires functional **or** performance enhancement. Software Maintenance/Support reapplies each of the preceding phases to an existing program.

Advantages:

- Simple and systematic.
- Linear ordering clearly marks the end of the one phase and starting of another phase
- The output of particular phase will be input put for next phase there for this output are normally referred as intermediate product or based line

Disadvantages:

- Real projects rarely follow the sequential flow that the model proposes. Changes can cause confusion as the project team proceeds.

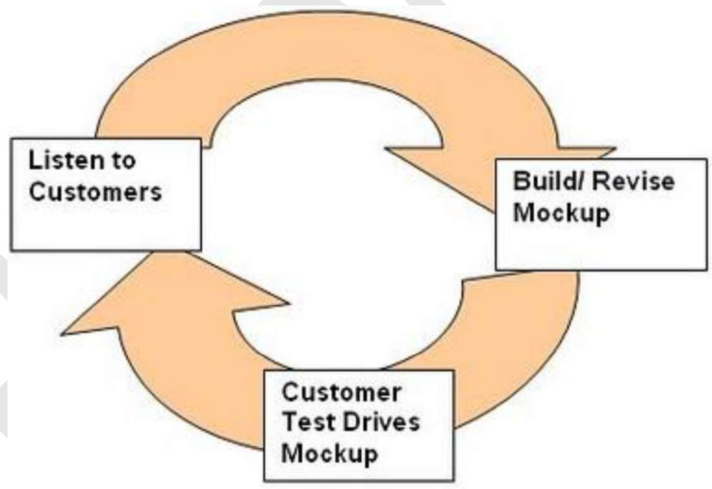
- It is difficult for the customer to state all requirements explicitly at the beginning of the projects.
- The water fall model assumes that the requirement should be completely specified before the rest of the development can proceed. In some situations it might be required that, first develop a part of the system completely and then later enhance a system where the client face an important role in requirement specification.
- The customer must have patience. A working version of program(s) will not be available until late in the project time span.
- Development is often delayed unnecessarily. The linear nature of the classic life cycle leads to "Blocking state" in which some project team members must wait for other members of the team to complete dependant tasks.
- The time spent waiting can exceed the time spent on productive work.

Prototyping Model

Many times following situations are occurred.

- Customers define a set of general objectives for software, but not detailed input, processing or output requirements.
- The developer may be unsure of the efficiency of an algorithm, the adaptability of an operation system, or the form that human-machine interaction should take.

In such cases prototyping approach can be used. Prototyping is an approach that enables the developer to create a model of the software that must be built.



The aim of this model is to overcome the limitations of the waterfall model that is instead of freezing the requirements before the design or coding phase a throw-away prototype is built to help understand the requirements. The prototyping model may be in any one of these forms:

1. **Paper prototype (PC based model):** It enables the user to understand how interaction will occur (Slide Show or diagram).
2. **Working prototype:** It implements some subset of the function required for the desired software.
3. **An existing program:** It performs all the existing function required but we can improve it for the new development.

- The prototyping paradigm begins with **requirements gathering**. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- A "**quick design**" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype.
- The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.
- Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make user of existing program fragments or applies tools. (e.g. report generation, window managers, etc…) that enable working program to be generated quickly.

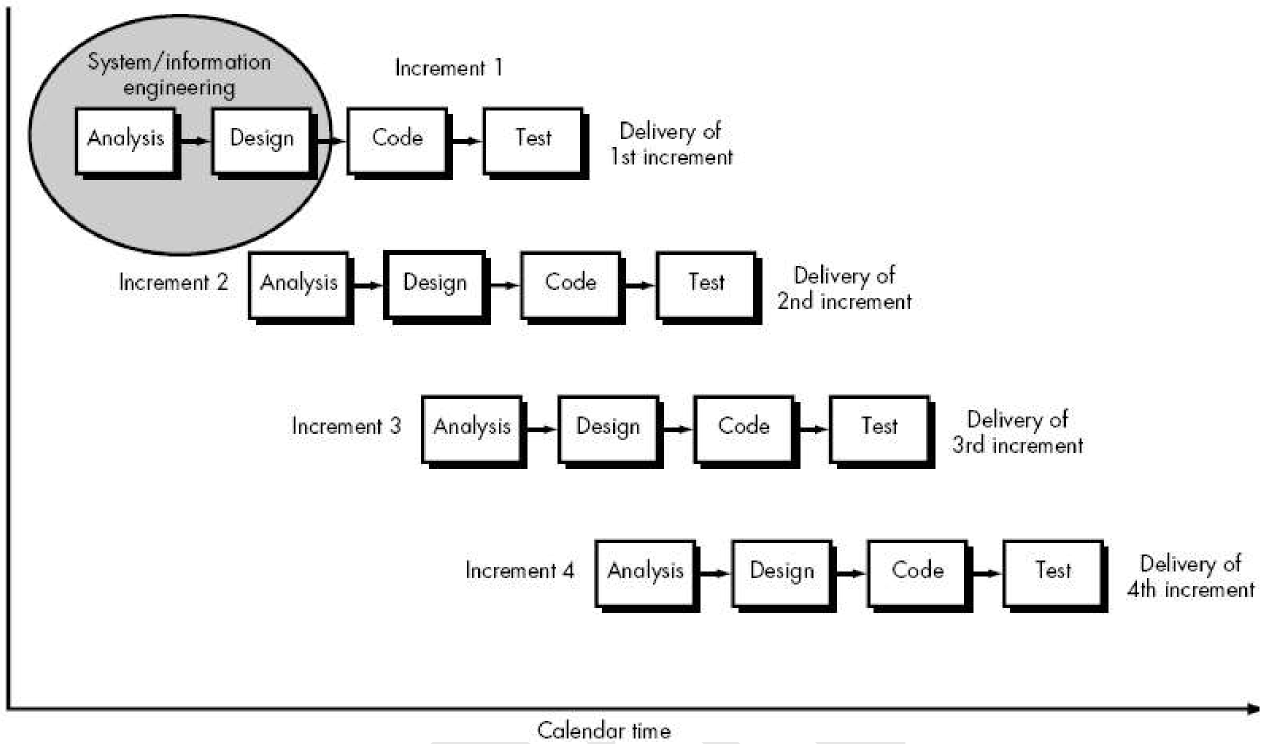
The important characteristics of Prototype Model are:

- The cost of requirement analysis must be kept low, inorder for it to be feasible.
- The development approach followed is "quick and dirty", the focus is on quicker development than on the quality.
- Only minimal documentation is required because it is a throw-away prototype model.
- This model is very useful where requirements are not properly understood in the beginning.
- It is an excellent method for reducing some types of risks involved with a project.

- The developer often makes implementation compromises in order to get a prototype working quickly.

Incremental model

- The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping.
- Referring to Figure, the incremental model applies linear sequences in a staggered fashion as calendar time progresses.



- Each linear sequence produces a deliverable "increment" of the software.
- For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.
- It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered.
- The core product is used by the customer (or undergoes detailed review).
- As a result of use and/or evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature.
- But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.
- Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.
- Early increments can be implemented with fewer people.
- If the core product is well received, then additional staff (if required) can be added to implement the next increment.
- In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of

this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

When to use Incremental models?

1. Requirements of the system are clearly understood
2. When demand for an early release of a product arises
3. When software engineering team are not very well skilled or trained
4. When high-risk features and goals are involved
5. Such methodology is more in use for web application and product based companies

Advantages:

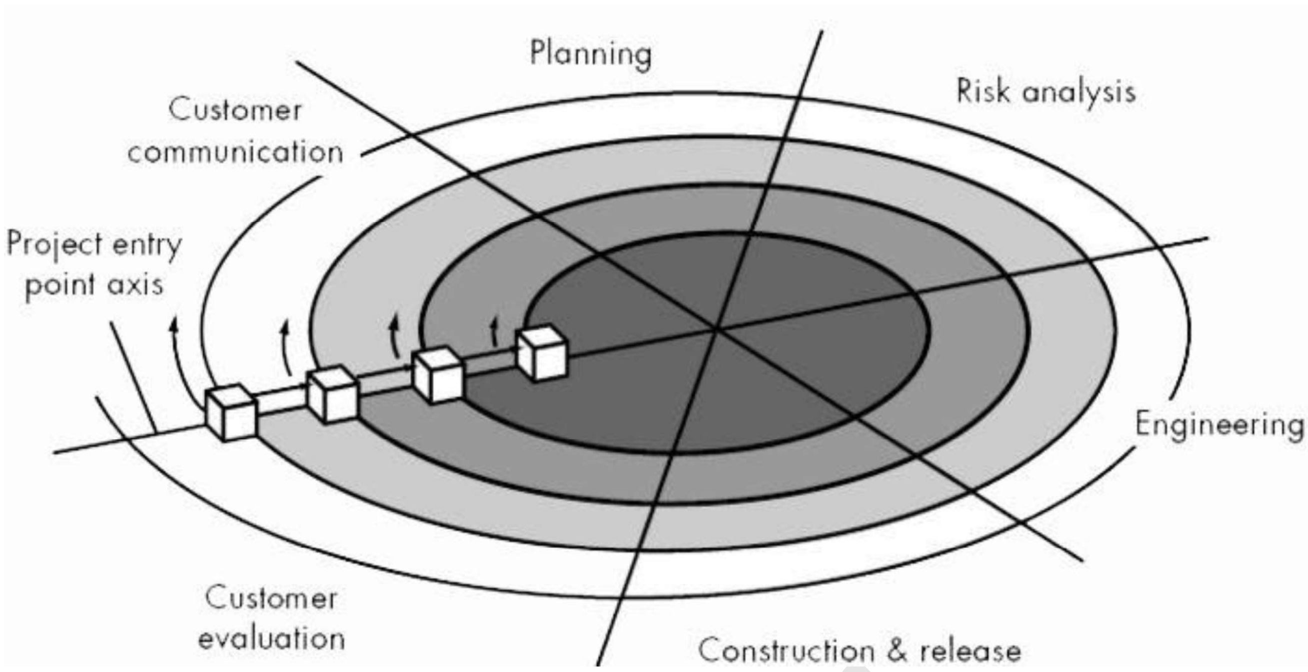
6. The software will be generated quickly during the software life cycle
7. It is flexible and less expensive to change requirements and scope
8. Throughout the development stages changes can be done.
9. This model is less costly compared to others.
10. A customer can respond to each increment.
11. Errors can be easily identified.

Disadvantages:

1. It requires a good planning designing.
2. Problems might cause due to system architecture as such not all requirements collected up front for the entire software lifecycle.
3. Each iteration phase is rigid and does not overlap each other.
4. Rectifying a problem in one unit requires correction in all the units and consumes a lot of time

Spiral Model

- The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model.
- It provides the potential for rapid development of incremental versions of the software.
- Using the spiral model, software is developed in a series of incremental releases.
- During early iterations, the incremental release might be a paper model or prototype.
- During later iterations, increasingly more complete versions of the engineered system are produced.
- A spiral model is divided into a number of framework activities, also called task regions.
- Typically, there are between three and six task regions. Figure depicts a spiral model that contains six task regions:
    - **Customer communication**: tasks required to establish effective communication between developer and customer.
    - **Planning:** tasks required to define resources, timelines, and other projectrelated information.
    - **Risk analysis**: tasks required to assess both technical and management risks.
    - **Engineering**: tasks required to build one or more representations of the application.
    - **Construction and release:** tasks required to construct, test, install, and provide user support (e.g., documentation and training).
    - **Customer evaluation**: tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.



- Each of the regions is populated by a set of work tasks, called a task set, that are adapted to the characteristics of the project to be undertaken.
- For small projects, the number of work tasks and their formality is low.
- For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality.
- In all cases, the umbrella activities (e.g., software configuration management and software quality assurance) are applied.
- As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center.
- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan.
- Cost and schedule are adjusted based on feedback derived from customer evaluation.
- In addition, the project manager adjusts the planned number of iterations required to complete the software.

Advantages:
- The spiral model is a realistic approach to the development of large-scale systems and software.
- Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.
- It uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at any stage in the evolution of the product.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
- It demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

Disadvantages:
- It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable.
- It demands considerable risk assessment expertise and relies on this expertise for success.
- If a major risk is not uncovered and managed, problems will undoubtedly occur.
- Finally, the model has not been used as widely as the linear sequential or prototyping paradigms.
- It will take a number of years before efficiency of this important paradigm can be determined with absolute certainty.

When to use Spiral Methodology?
1. When project is large
2. When releases are required to be frequent
3. When creation of a prototype is applicable

4. When risk and costs evaluation is important
5. For medium to high-risk projects
6. When requirements are unclear and complex
7. When changes may require at any time
8. When long term project commitment is not feasible due to changes in economic priorities

4GT Techniques

The term 4GT means a broad array of software tools that have one thing in common i.e. each enables the software engineer to specify some characteristic of software at a high level. The tool then automatically generates source codes base on the development specification. Currently, the software development environment that supports the 4GT paradigm includes some or all of the following tools:

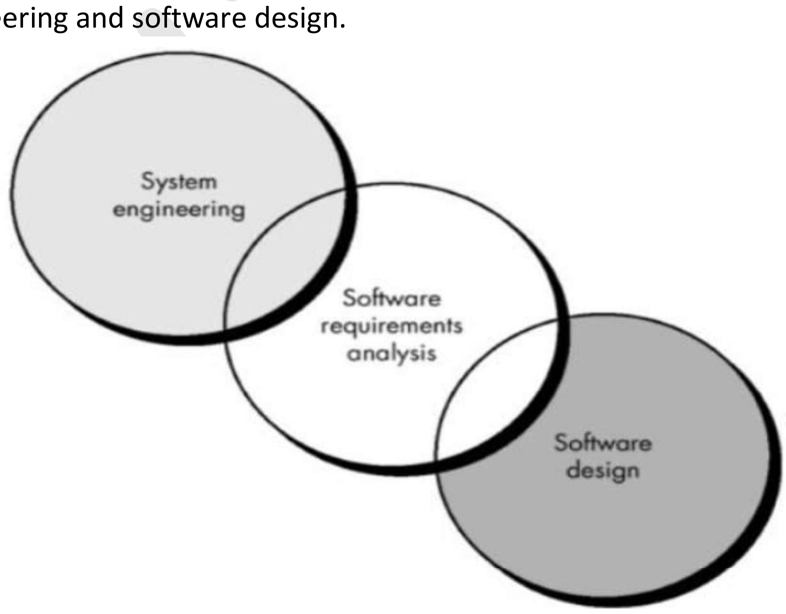
- Non procedural languages for database query.
- Report generation.
- Screen interaction & definition.
- Code generation.
- Graphics at high level capabilities, spreadsheet capability and automated generation of html and similar languages used for website creation using advanced software tools.

Implementation using a 4GT enables the software developers to represent desired results in a manner that leads to automatic generation of code to create those results. To transform a 4GT implementation into a product, the developer must conduct through testing or details testing, develop, meaningful documentation and perform all other solution integration activities that are required other software engineering. 4GT tools are not all that much easier to use than programming language and result source code produce by such tools is inefficient and the maintenance of large software system becomes most difficult. There are some advantages for using 4GT.

1. The use of 4GT is a viable approach for many different applications are as for smaller application it is a better solution to use.
2. Data collected for companies that use 4GT indicate that require to produce software is quietly reduced and intermediate applications and that the amount of design and analysis for small application is also reduce.
3. The use of 4GT for large software development efforts demands as much or more analysis, design and testing to achieve substantial time saving that reduce for the elimination of coding.

Requirement Analysis

- *Requirements analysis* is a software engineering task that bridges the gap between system level requirements engineering and software design.



- Requirements analysis provides the software designer with a representation of information, function, and behaviour that can be translated to data, architectural, interface, and component-level designs.
- It also indicates software's interface with other system elements, and establish constraints that software must meet.